Harnessing the Power of OpenCL

Code in motion

Session 308

These are confidential sessions—please refrain from streaming, blogging, or taking pictures.

Harnessing the Power of OpenCL

Ian Ollmann, Ph.D.
Senior Scientist

Goals

- Understand OpenCL in the context of other frameworks
  - Box filter
- Explore performance tuning with OpenCL APIs
  - Conversions conformance test
- Moving to OpenCL
  - Cloth simulation (Andrew Brownsword—EA)

Duarte:
Third major bullet point title should match the title on Andrew Brownsword's talk also in this session,

- Done (kh)

OpenCL in the Context of Other Frameworks

Box filter

OpenCL in Context

- Other similar tool chains
  - pthreads + GCC
    - CPU only
    - Predictable arithmetic results
    - Very flexible—Use any algorithm

Example:
framework A

  - OpenGL + shader compiler
    - CPU + GPU
    - Arithmetic is hardware dependent
    - Less flexible—Must present computation as a graphics operation

Example:
framework B

Example: Box Filter

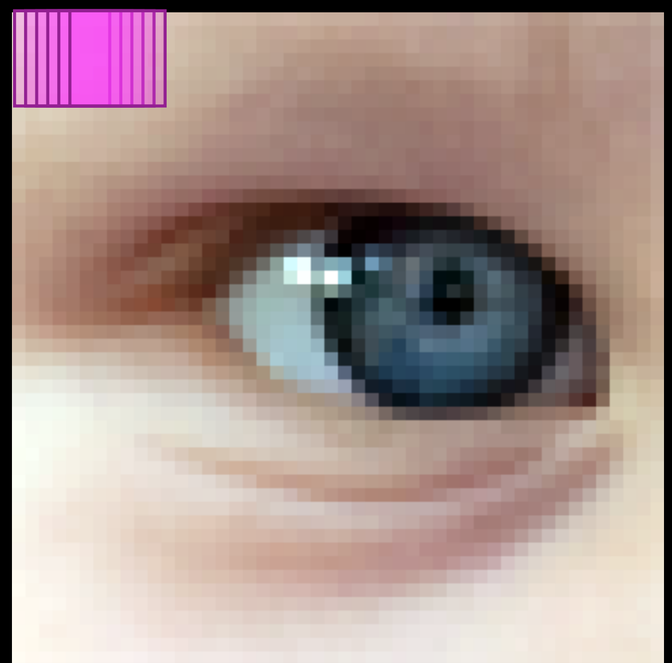
Duarte:

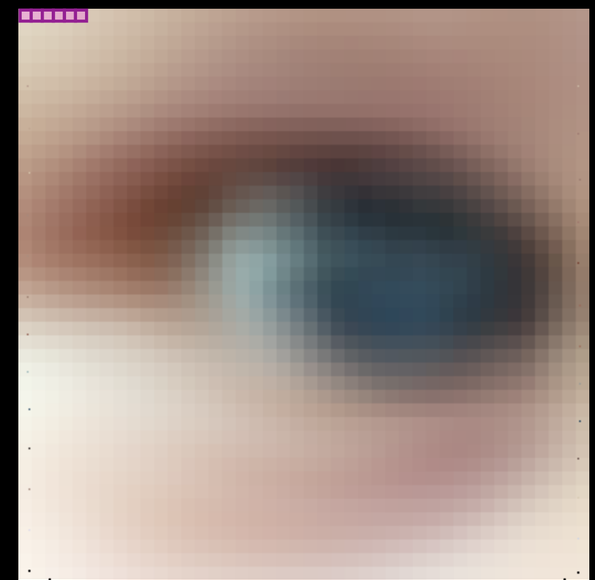
This slide should contain similar image content as the similar slide in session 307.

- Done (kh)

There are some JPEG artifacts here that might be nice toclean up.

- A simple average over a rectangular region of adjacent pixels





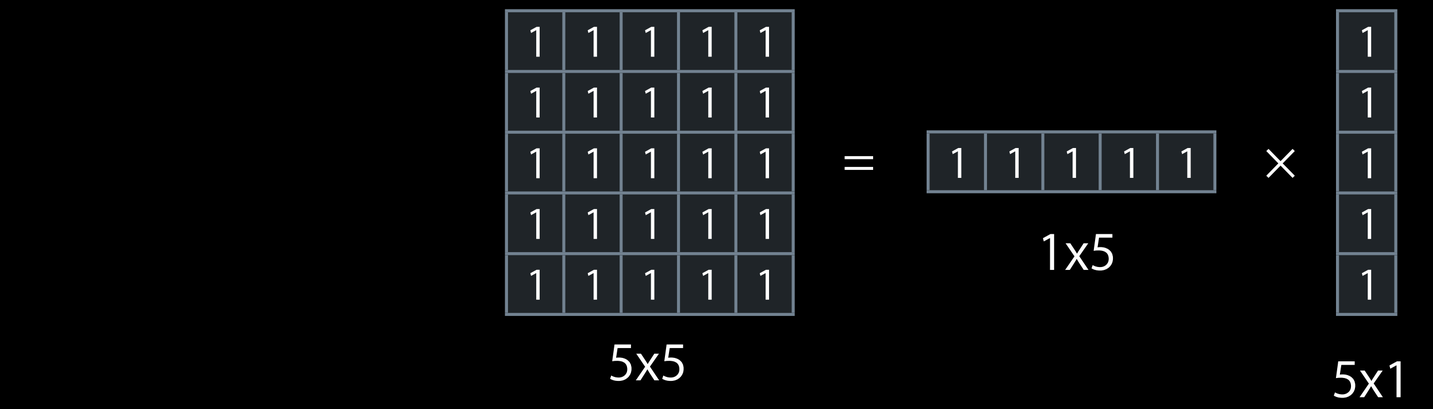
Box Filter: Sample Kernel

```
kernel void box_filter( read_only image2d_t   src,
                        write_only image2d_t  dest,
                                       int  radius  )
{
  const sampler_t sampler = CLK_ADDRESS_CLAMP_TO_EDGE | CLK_FILTER_NEAREST;
  int destX = (int) get_global_id(0);    // The x coordinate of the pixel we intend to produce
  int destY = (int) get_global_id(1);    // The y coordinate of the pixel we intend to produce
  int i, j;

  // Sum over all pixels in the adjacent area
  float4 sum = 0.0f;
  for( i = -radius; i <= radius; i++ )
     for( j = -radius; j <= radius; j++ )
        sum += read_imagef( src, sampler, (int2) ( destX + j, destY + i ) );

  // Divide the result by the number of pixels in the area
  float pixel_count_1D = 2 * radius + 1;
  sum /= pixel_count_1D * pixel_count_1D;

  // write out the result
  write_imagef( dest, (int2)( destX, destY ), sum );
}
```

Box Demo

Always Use the Right Algorithm

- Framework B uses separable filters



Cost:

24 adds/pixel

8 adds/pixel

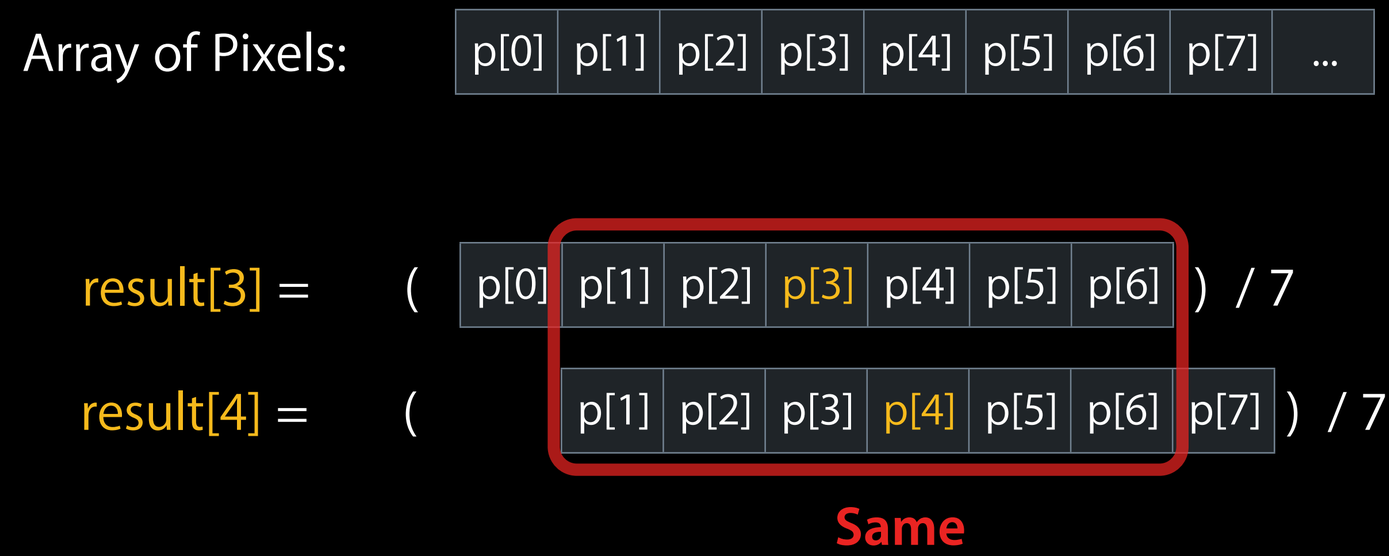
Box Filter: Separated Kernels

```
kernel void kernel2_horizontal( read_only image2d_t     src,
                                write_only image2d_t  dest,
                                int                       radius,
                                float                     reciprocal   )
{
   const sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_CLAMP_TO_EDGE | CLK_FILTER_NEAREST;
   int destX = (int) get_global_id(0);     // The x coordinate of the pixel we intend to produce
   int destY = (int) get_global_id(1);     // The y coordinate of the pixel we intend to produce
   int i, j;

   // Sum over all pixels in the adjacent area
   float4 sum = 0.0f;
   for( i = -radius; i <= radius; i++ )
        sum += read_imagef( src, sampler, (int2) ( destX + i, destY ) );

   // Divide the result by the number of pixels in the area
   sum *= reciprocal;

   // write out the result
   write_imagef( dest, (int2)( destX, destY ), sum );
}
```

```
kernel void kernel2_vertical( read_only image2d_t       src,
                              write_only image2d_t  dest,
                              int                       radius,
                              float                  reciprocal   )
{
   const sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_CLAMP_TO_EDGE | CLK_FILTER_NEAREST;
   int destX = (int) get_global_id(0);     // The x coordinate of the pixel we intend to produce
   int destY = (int) get_global_id(1);     // The y coordinate of the pixel we intend to produce
   int i, j;

   // Sum over all pixels in the adjacent area
   float4 sum = 0.0f;
   for( i = -radius; i <= radius; i++ )
        sum += read_imagef( src, sampler, (int2) ( destX , destY + i ) );

   // Divide the result by the number of pixels in the area
   sum *= reciprocal;

   // write out the result
   write_imagef( dest, (int2)( destX, destY ), sum );
}
```

Box Demo

Always Use the Best Algorithm



$\therefore$ result[4] = result[3] + (p[7] - p[0]) / 7

Constant cost, regardless of filter radius!

OpenGL Shader Languages Cannot Do This

Box Demo

Performance Tuning with OpenCL APIs

Conversions conformance test

The Problem: Testing Conversions

```
InType y;
OutType  x = convert_typen<_sat><_rnd>( y );
```

10 input types — char, uchar, short, ushort...

10 output types — char, uchar, short, ushort...

5 vector sizes — scalar, type2, type4, type8, type16

5 rounding modes — nearest, up, down, zero, current

× 2 saturation modes — "_sat" and ""

5000 conversions to write / test

× 2.6 billion values / conversion (on average)

Testing Philosophy: "Test every number where possible"

12 trillion values to test!

The Answer to the Question You Really Wanted Answered

What could possibly be interesting about a conformance test?

It has to finish

Conversions Test—Basic Algorithm

Starting algorithm:

Init a series of numbers to be tested
clEnqueueWriteBuffer(blocking) to copy it to the device
For each vector size:
clEnqueueWriteBuffer (blocking) to overwrite *output* buffer with garbage
clEnqueueNDRangeKernel:

```
kernel void foo( global uint *src, global int *dest )
{
   size_t i = get_global_id(0);
   dest[i] = convert_uint( src[i] );
}
```

clFlush() to get OpenCL moving
Calculate the reference answer in the main thread
For each vector size:
clEnqueueReadBuffer(blocking) to read the results back
if(memcmp(results, reference, size))
Report an error

Conversions Test—Performance

Starting performance:

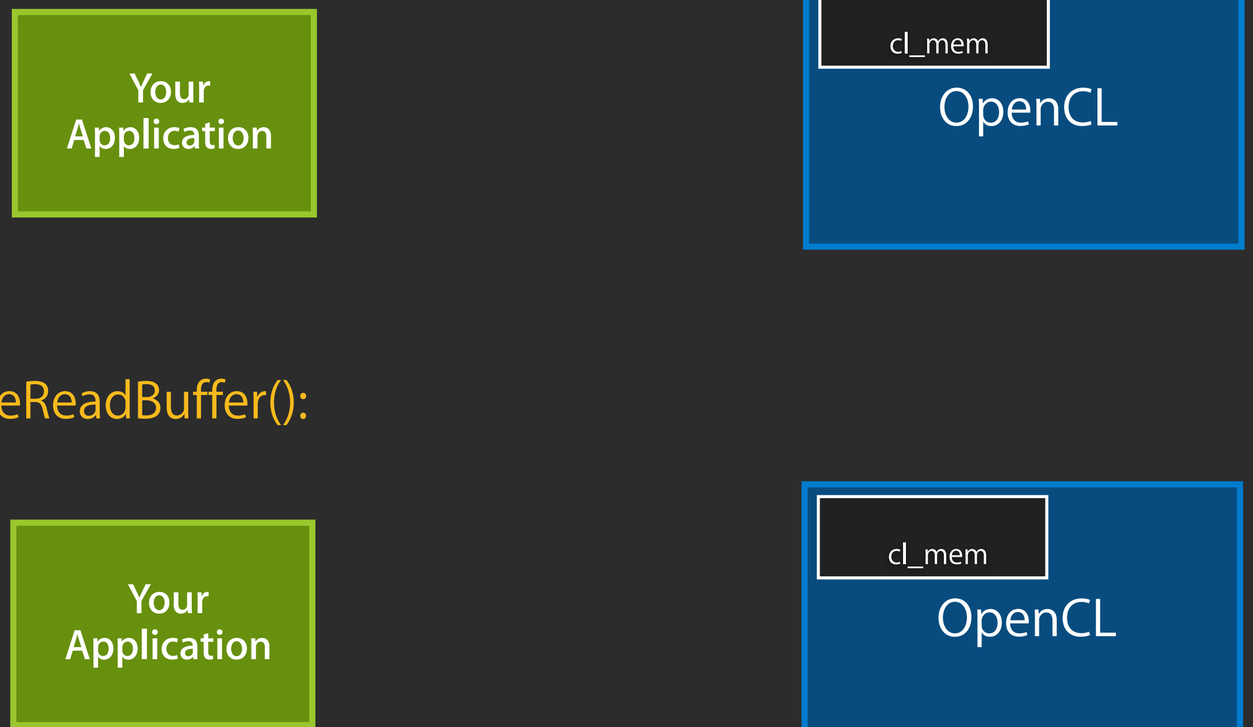
	GPU	Four-Core CPU
Start	242 s	230 s

Shark:

!	Self ▼	Total	Library	Symbol
	31.3%	31.3%	libSystem.B.dylib	▶ __longcopy
	17.0%	17.0%	libcldcpuengine.d…	▶ cld_exec_workitem
	14.4%	14.4%	libSystem.B.dylib	▶ __memset_pattern
	9.0%	9.0%	libSystem.B.dylib	▶ memcmp
!	5.7%	5.7%	Unknown Library	▶ 0x1003f12c0 [70B]
	4.0%	4.0%	test_conversions1	▶ uint2int_many

clEnqueueRead/WriteBuffer

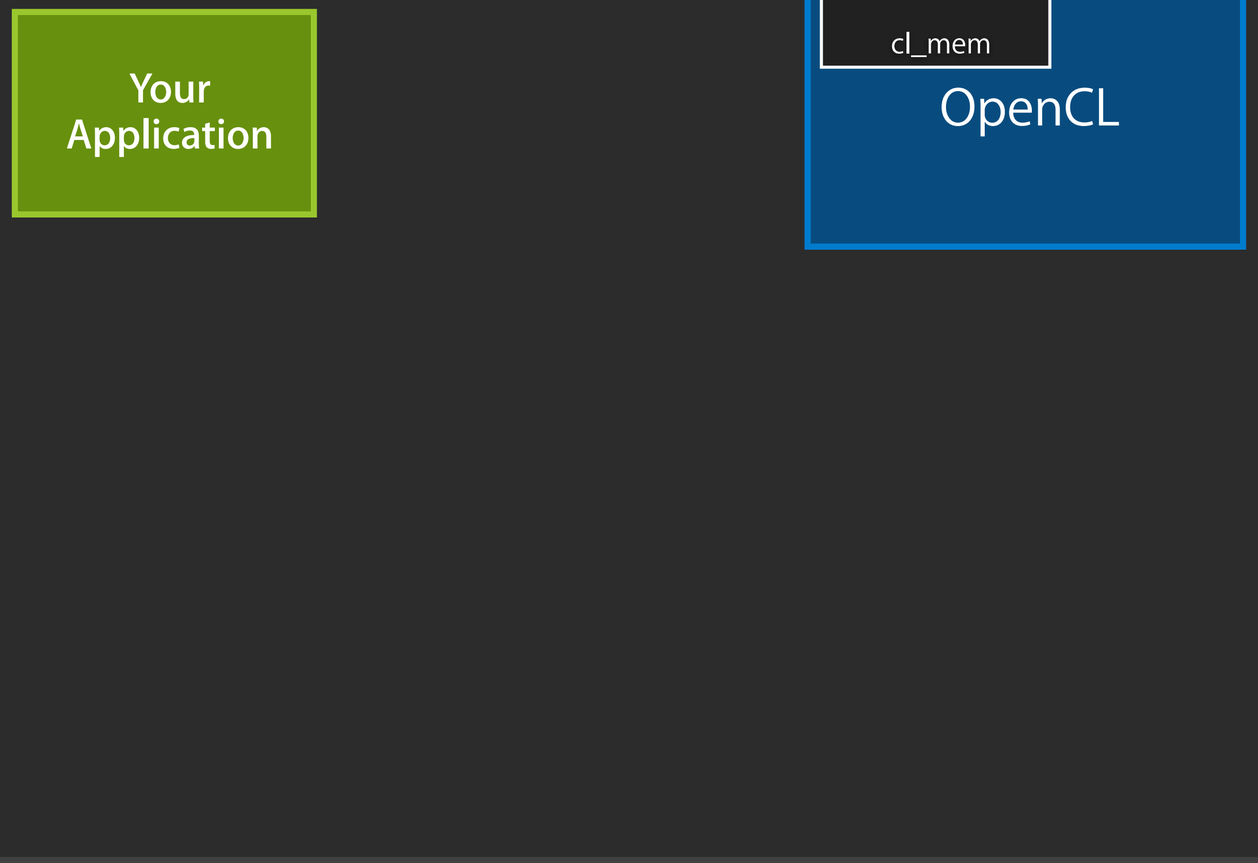
clEnqueueWriteBuffer():



clEnqueueReadBuffer():

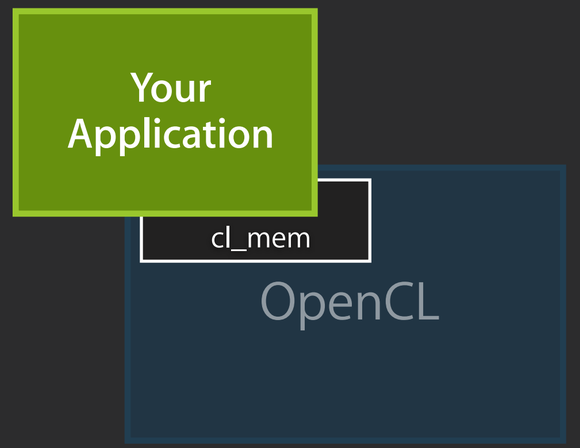
clEnqueueMapBuffer

clEnqueueMapBuffer():



clEnqueueMapBuffer

clEnqueueUnmapMemObject():



Conversions Test—Improved

Code:

Init the input data
clEnqueueWriteBuffer(blocking) to copy it to the device
For each vector size:
clEnqueueWriteBuffer (blocking) to overwrite output buffer with garbage
clEnqueueNDRangeKernel:

```
kernel void foo( global uint *src, global int *dest )
{
   size_t i = get_global_id(0);
   dest[i] = convert_uint( src[i] );
}
```

clFlush() to get OpenCL moving
Calculate the reference answer in the main thread
For each vector size:
clEnqueueReadBuffer(blocking) to read the results back
if(memcmp(results, reference, size))
Report an error

Conversions Test—Improved

Code:

Init the input data
clEnqueueWriteBuffer(blocking) to copy it to the device
For each vector size:
clEnqueueNDRangeKernel:

```
kernel void foo( global uint *src, global int *dest )
{
   size_t i = get_global_id(0);
   dest[i] = convert_uint( src[i] );
}
```

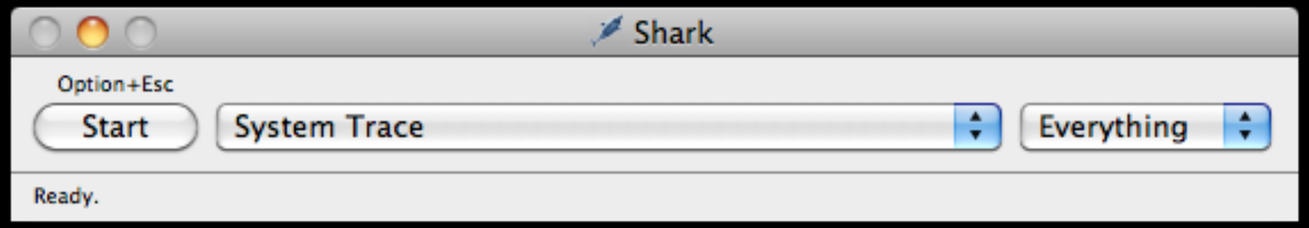
clFlush() to get OpenCL moving
Calculate the reference answer in the main thread
For each vector size:
clEnqueueMapBuffer(blocking) to read the results back
if(memcmp(results, reference, size))
Report an error
Fill results buffer with garbage
clEnqueueUnmapMemObject() to release the mapped buffer

Conversions Test—Performance

Optimized performance:

	GPU	Four-Core CPU
Start	242 s	230 s
Use Map	173 s (40%)	128 s (80%)

Shark:

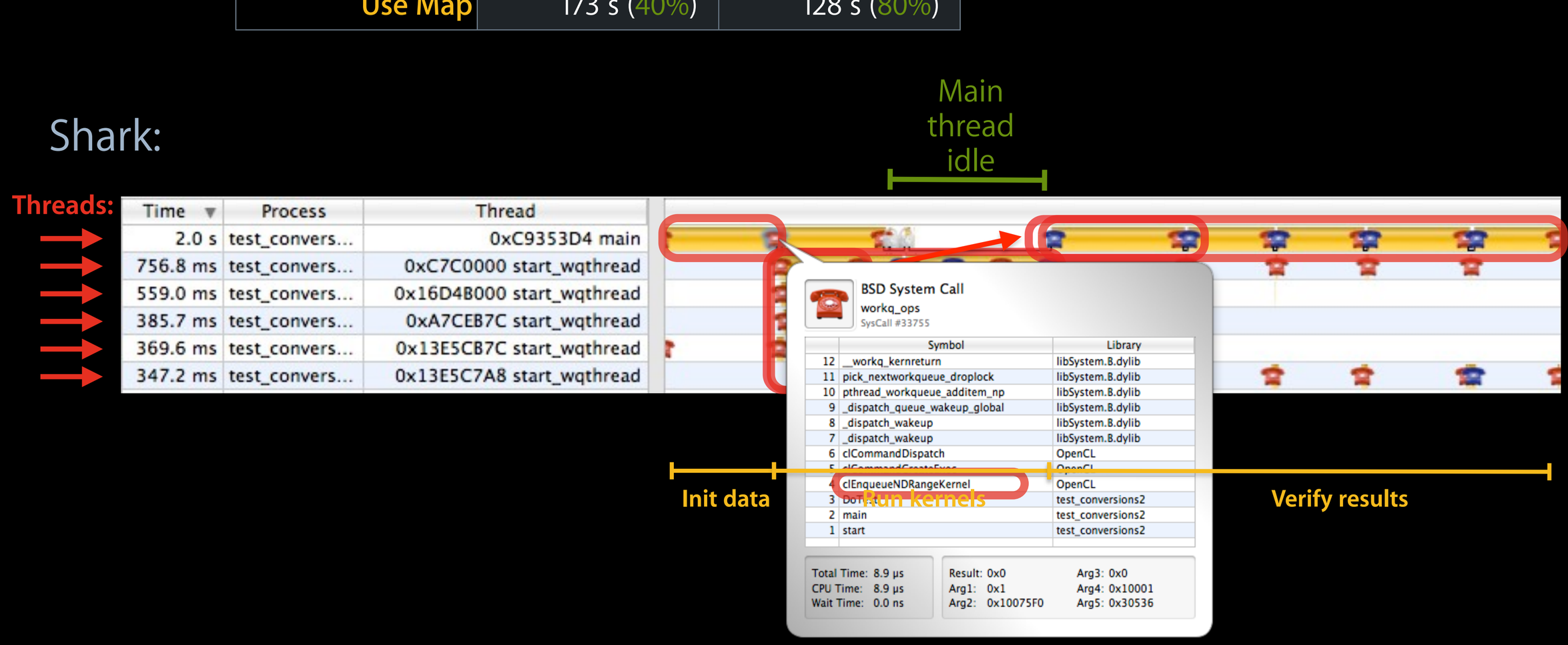


Conversions Test—Performance

Using clEnqueueMapBuffer:

	GPU	Four-Core CPU
Start	242 s	230 s
Use Map	173 s (40%)	128 s (80%)

Shark:



Conversions Test—Improved Algorithm

Code:

Init the input data

clEnqueueWriteBuffer(blocking) to copy it to the device

For each vector size:

clEnqueueNDRangeKernel:

```
kernel void foo( global uint *src, global int *dest )
{
   size_t i = get_global_id(0);
   dest[i] = convert_uint( src[i] );
}
```

clFlush() to get OpenCL moving

Calculate the reference answer in the main thread

For each vector size:

clEnqueueMapBuffer(non-blocking) to map the results buffer

clWaitForEvents() to read the results back

if(memcmp(results, reference, size))

Report an error

Fill results buffer with garbage

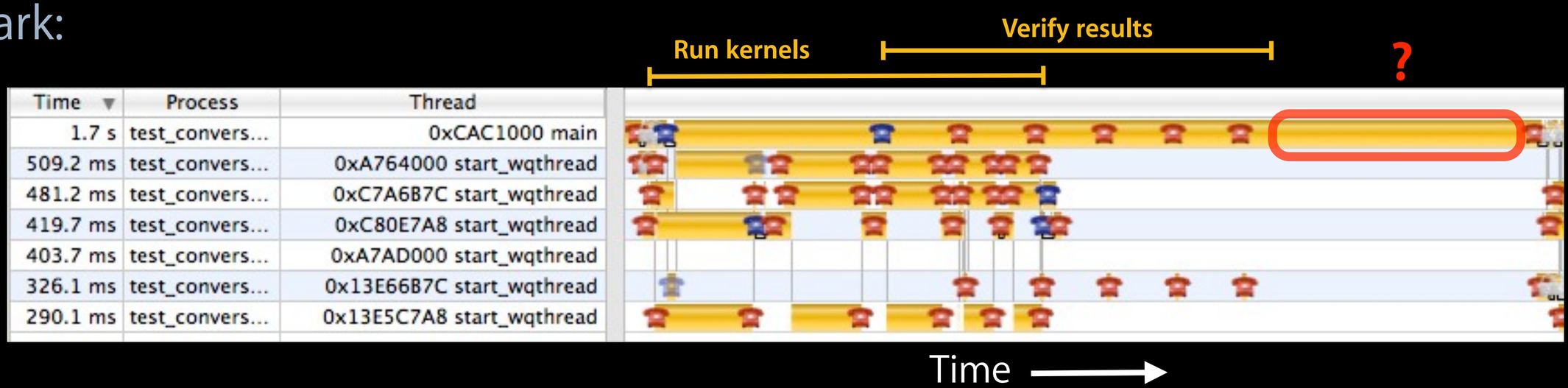
clEnqueueUnmapMemObject() to release the mapped buffer

Conversions Test—Performance

Nonblocking map performance:

	GPU	Four-Core CPU
Start	242 s	230 s
Use Map	173 s (40%)	128 s (80%)
Nonblocking Map	174 s (39%)	119 s (93%)

Shark:



Conversions Test—Remove Blocking Write

Optimized code:

clEnqueueMapBuffer(blocking)

Init the input data

clEnqueueWriteBuffer(non-blocking) to copy it to the device

clEnqueueUnmapMemObject()

For each vector size:

clEnqueueNDRangeKernel:

```
kernel void foo( global uint *src, global int *dest )
{
   size_t i = get_global_id(0);
   dest[i] = convert_uint( src[i] );
}
```

clEnqueueMapBuffer(non-blocking) to read the results back

clFlush() to get OpenCL moving

Calculate the reference answer in the main thread

For each vector size:

clWaitForEvents(1, map event) to make sure the map is done

if(memcmp(results, reference, size))

Report an error

Fill results buffer with garbage

clEnqueueUnmapMemObject() to release the mapped buffer

Conversions Test—Remove Blocking Write

Remove blocking write performance:

	GPU	Four-Core CPU
Start	242 s	230 s
Use Map	173 s (40%)	128 s (80%)
Nonblocking Map	174 s (39%)	119 s (93%)
Remove Blocking Write	158 s (53%)	122 s (88%)

Why not much improvement?

!	Self	Total	Library	Symbol
	18.6%	18.6%	libcldcpuengine.d...	cld_exec_workitem
	11.2%	11.2%	libSystem.B.dylib	__memset_pattern
!	9.6%	9.6%	Unknown Library	0x1000e52c0 [87B]
	8.6%	8.6%	test_conversions3	init_uint
	7.9%	7.9%	libSystem.B.dylib	memcmp
!	6.8%	6.8%	Unknown Library	0x1000e62f0 [89B]
	6.3%	6.3%	test_conversions3	uint2uchar_many
	5.6%	5.6%	libSystem.B.dylib	__longcopy

Conversions Test—Remove Blocking Write

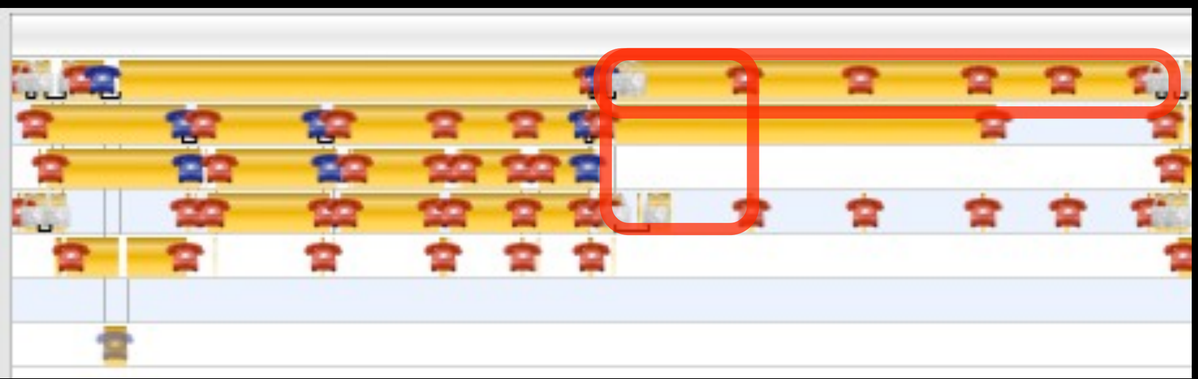
Remove blocking write performance:

	GPU	Four-Core CPU
Start	242 s	230 s
Use Map	173 s (40%)	128 s (80%)
Nonblocking Map	174 s (39%)	119 s (93%)
Remove Blocking Write	158 s (53%)	122 s (88%)

Shark:

Poor CPU utilization

Time	Process	Thread
2.8 s	test_convers...	0xA7113D4 main
2.1 s	test_convers...	0x9E377A8 start_wqthread
676.6 ms	test_convers...	0x13E6B7A8 start_wqthread
558.4 ms	test_convers...	0x13E5B7A8 start_wqthread
557.8 ms	test_convers...	0xCAC1000 Idle Thread - CPU1
498.9 ms	test_convers...	0x9E39000 start_wqthread
275.9 ms	test_convers...	0x13E5A000 start_wqthread



Run kernels

Verify results

Conversions Test—Use OpenCL to Verify

Optimized code:

clEnqueueMapBuffer(blocking)

Init the input data, clEnqueueUnmapMemObject()

For each vector size:

clEnqueueNDRangeKernel:

```
kernel void foo( global uint *src, global int *dest )
{
   size_t i = get_global_id(0);
   dest[i] = convert_uint( src[i] );
}
```

clEnqueueNDRangeKernel:

Add new Kernel to check results

```
kernel void checkByte( global uchar *src1, global uchar *src2, global int *err )
{
   size_t i = get_global_id(0);
   if( src1[i] != src2[i] ){ *err = i; return; };
}
```

clFlush() to get OpenCL moving

Calculate the reference answer in the main thread for next loop

For each vector size:

Check to make sure no error was returned

Conversions Test—Performance

Use OpenCL to verify:

	GPU	Four-Core CPU
Start	242 s	230 s
Use Map	173 s (40%)	128 s (80%)
Nonblocking Map	174 s (39%)	119 s (93%)
Remove Blocking Write	158 s (53%)	122 s (88%)
Use OpenCL to Verify	70 s (3.5x)	105 s (2.2x)

Shark:

Longer than expected

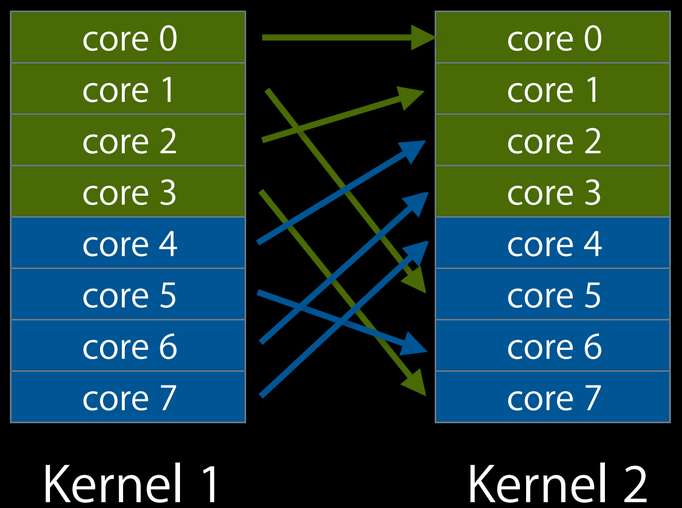
Time	Process	Thread
2.7 s	test_convers...	0x96F23D4 start_wqthread
1.7 s	test_convers...	0x13E1A7A8 main
1.3 s	test_convers...	0x16D4B3D4 start_wqthread
949.9 ms	test_convers...	0x13E6B000 Idle Thread - CPU1
830.2 ms	test_convers...	0xA764000 start_wqthread
775.6 ms	test_convers...	0x956CB7C start_wqthread

Conversions Test—Performance

- What happened?
  - Before
    - memcmp()—Hand tuned, vectorized assembly code
  - After
    - my memcmp kernel—Lazily written, byte-at-a-time compare

- Also

CPU:



GPU: Extra trip to global memory and back

Conversions Test—Fuse Kernels

Optimized code:

clEnqueueMapBuffer(blocking)

Init the input data, clEnqueueUnmapMemObject()

For each vector size:

clEnqueueNDRangeKernel:

Fuse two kernels into one

```
kernel void foo( global uint *src, global int *ref, global int *err )
{
   size_t i = get_global_id(0);
   if( ref[i] != convert_uint( src[i] ) )
   {
      *err = i;
      return;
   }
}
```

clFlush() to get OpenCL moving

Calculate the reference answer in the main thread for next loop

For each vector size:

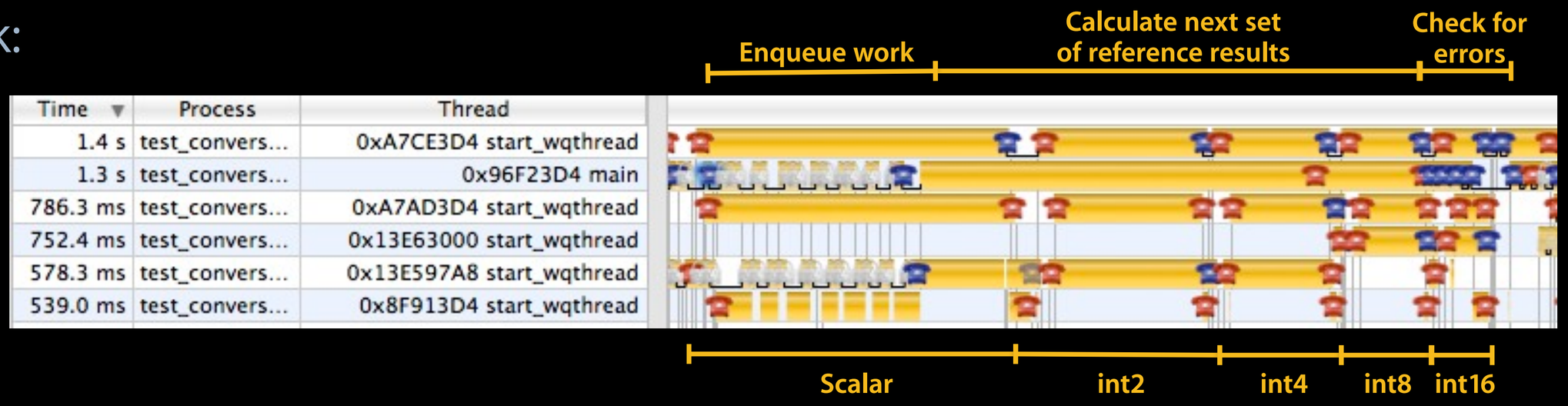
Check to make sure no error was returned

Conversions Test—Performance

Fuse kernels for testing and verification:

	GPU	Four-Core CPU
Start	242 s	230 s
Use Map	173 s (40%)	128 s (80%)
Nonblocking Map	174 s (39%)	119 s (93%)
Remove Blocking Write	158 s (53%)	122 s (88%)
Use OpenCL to Verify	70 s (3.5x)	105 s (2.2x)
Fuse Test and Verify Kernels	64 s (3.8x)	42 s (5.5x)

Shark:



Are the Kernels Too Small?

Shark time profile:

!	Self ▼	Total	Library	Symbol
!	20.6%	20.6%	Unknown Library	▶ 0x1000fa380 [51B]
	13.7%	13.7%	a.out	▶ init_uint
	12.2%	12.2%	libcldcpuengine.d...	▶ cld_exec_workitem
!	12.2%	12.2%	Unknown Library	▶ 0x1000fb380 [97B]
	11.8%	11.8%	a.out	▶ uint2int
	7.4%	7.4%	a.out	▶ uint2int_many
!	6.5%	6.5%	Unknown Library	▶ 0x1000fc380 [83B]
!	3.6%	3.6%	Unknown Library	▶ 0x1000fd380 [115B]
!	3.2%	3.2%	mach_kernel	▶ ml_set_interrupts_enabled
!	2.7%	2.7%	Unknown Library	▶ 0x1000fe380 [179B]
	1.5%	1.5%	libSystem.B.dylib	▶ __memcpy

Too much time spent in cld_exec_workitem()

Disassembly for scalar kernel:

Self	Address ▲	Code	
9.1%	0x1000fa380	pushq	%rbp
	0x1000fa381	movq	%rsp, %rbp
	0x1000fa384	movq	%gs:344(), %rax
6.9%	0x1000fa38d	movq	(%rax), %rax
12.8%	0x1000fa391	movq	(%rdi), %rcx
0.4%	0x1000fa395	movl	(%rcx, %rax, 4), %ecx
52.6%	0x1000fa398	movq	+8(%rdi), %rdx
0.4%	0x1000fa39d	cmpl	(%rdx, %rax, 4), %ecx
4.5%	0x1000fa3a0	jz	0x00000001000fa3b1 <0x1000fa380 [51B]+49>
	0x1000fa3a6	movq	+16(%rdi), %rsi
	0x1000fa3ab	movl	%eax, (%rsi)
	0x1000fa3ae	movl	%ecx, (%rdx, %rax, 4)
13.3%	0x1000fa3b1	popq	%rbp
	0x1000fa3b2	ret	

Kernels are very small

Combine Everything in One Kernel!

```
kernel void test( global in_type *src, global out_type *ref, global int *err )
{
   size_t i = get_global_id(0);
   size_t j, k, l, m;
   out_type16 v16 = vload16( i, src );
   out_type16 ref16 = vload16( i, ref );
   out_type16 out16 = convert_type16( v16 );
   if( any( as_utype16(out16) != as_utype16( ref16 )) ) { err[4] = i; vstore16( out16, i, ref); return; }

   in_type8 v8[2] = { v16.lo, v16.hi };
   out_type8 ref8[2] = { ref16.lo, ref16.hi };
   out_type8 out8[2];
   for( j = 0; j < 2; j++ )
   {
       out8[j] = convert_out_type8( v8[j] );
       if( any( as_utype8(out8[j]) != as_utype8( ref8[j] )) ){ err[3] = 2*i+j; vstore8( out8[j], 2*i+j, ref); return; }

       in_type4 v4[2] = { v8[j].lo, v8[j].hi };
       out_type4 ref4[2] = { ref8[j].lo, ref8[j].hi };
       out_type4 out4[2];
       for( k = 0; k < 2; k++ )
       {
           out4[k] = convert_out_type4( v4[k] );
           if( any( as_utype4(out4[k]) != as_utype4( ref4[k] )) ) { err[2] = 4*i+2*j+k; vstore4( out4[k], 4*i+2*j+k, ref); return; }

           in_type2 v2[2] = { v4[k].lo, v4[k].hi };
           out_type2 ref2[2] = { ref4[k].lo, ref4[k].hi };
           out_type2 out2[2];
           for( l = 0; l < 2; l++ )
           {
               out2[l] = convert_out_type2( v2[l] );
               if( any( as_utype2(out2[l]) != as_utype2( ref2[l] )) ) { err[1] = 8*i+4*j+2*k+l; vstore2( out2[l], 8*i+4*j+2*k+l, ref); return; }

               in_type v1[2] = { v2[l].lo, v2[l].hi };
               out_type ref1[2] = { ref2[l].lo, ref2[l].hi };
               out_type out1[2];
               for( m = 0; m < 2; m++ )
               {
                   out1[m] = convert_out_type( v1[m] );
                   if( as_utype(out1[m]) != as_utype( ref1[m] )) { err[0] = 16*i+8*j+4*k+2*l+m; ref[16*i+8*j+4*k+2*l+m] = out1[m]; return; }
               }
           }
       }
   }
}
```

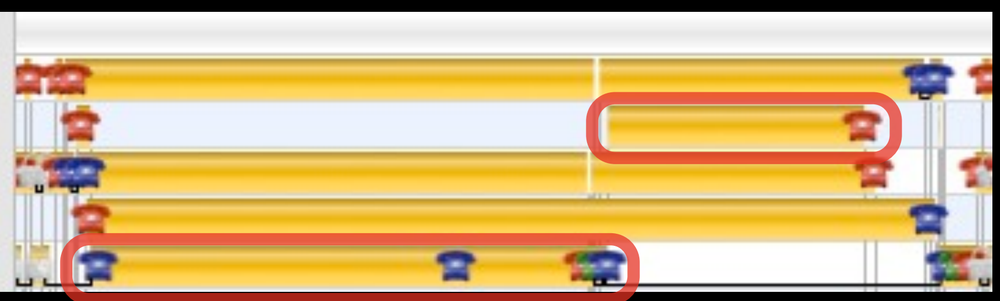
Conversions Test

Fuse Kernels for different vector sizes:

	GPU	Four-Core CPU
Start	242 s	230 s
Use Map	173 s (40%)	128 s (80%)
Nonblocking Map	174 s (39%)	119 s (93%)
Remove Blocking Write	158 s (53%)	122 s (88%)
Use OpenCL to Verify	70 s (3.5x)	105 s (2.2x)
Fuse Test and Verify Kernels	64 s (3.8x)	42 s (5.5x)
Fuse All Kernels	35 s (6.9x)	25 s (9.2x)

Shark:

Time	Process	Thread
2.2 s	test_convers...	0xC7C0000 start_wqthread
2.2 s	test_convers...	0x13E597A8 start_wqthread
1.7 s	test_convers...	0x13E1A000 start_wqthread
1.4 s	test_convers...	0x952D000 start_wqthread
1.2 s	test_convers...	0x9E39000 main



Calc next set of input data + results

Help finish current task

Conversions Test

- Findings
  - Unsuccessful strategies
    - Moving work around to fill “holes” in Shark System Traces
    - Replacing vectorized single threaded app code with unvectorized OpenCL code
    - Ignoring the Shark Time Profile
  - Successful strategies
    - Eliminating copies and other work
    - Moving single thread application work to OpenCL kernels (well written)
    - Merging multiple kernels into one to reduce global memory accesses

Vectorization vs. Multithreading

- You can do both!
  - With OpenCL, multithreading "just happens"
  - You still have to write the vector code
    - Much easier—shared consistent operators for scalar + vector
    - Portable!

Vectorization vs. Multithreading

- Multithreading is not a replacement for vectorization
  - Using more cores consumes more power
    - Less battery life
    - Less processing power available for other apps

- Vectorizing code saves power
  - Many fewer instructions to decode
  - More regular memory access

Vectorization vs. Multithreading

- Using the GPU can save power too
  - Processing elements are much smaller and cheaper
  - Lower clock
- Vectorizing can make the GPU run faster too
  - Bigger types for coalesced memory accesses
  - More parallel work can help with compiler scheduling (but may increase resources used per workitem)

Moving to OpenCL

Andrew Brownsword
Chief Architect
Electronic Arts BlackBox

Overview

- Motivation
- Tasking model
- Data model
- Kernels
- Concurrency
- Performance

Motivation

Motivation for OpenCL

- Supports a wide variety of compute resources

- Open standard

- Concurrent programming model

Motivation for OpenCL

- Supports a wide variety of compute resources

- Open standard

- Concurrent programming model

Motivation for OpenCL

- Supports a wide variety of compute resources
- Open standard
- Concurrent programming model

A Demo Is Born

- Goal: Exercise OpenCL on “real” code
  - Extract game subsystem and port key algorithms to OpenCL
  - Leave intact as much original code and API as possible

- Character skinning and cloth physics from EA’s Skate 2

What Does It Do?

- Play back recorded skeleton poses

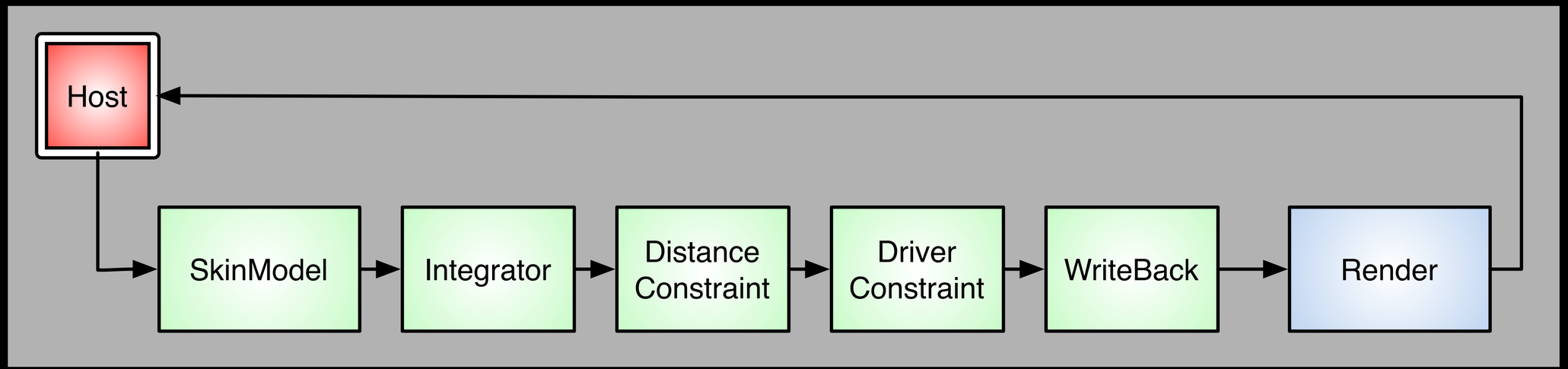
- Skin character model to pose

- Apply cloth physics to portion of the skinned model
  - Integrator for gravity
  - Particle-to-particle spring system
  - Constraint to underlying skinned model

- Render via OpenGL

Tasking Model

Demo Task Graph



- Simple sequential execution graph
- In-order, data parallel tasks
- Render inputs are double buffered to OpenCL tasks could, in theory, begin prior to render completion

Enqueue Instead of Execute

- In original code each box represents a function call

- Demo abstracts OpenCL kernel invocations as functors
  - Functors take an event parameter to be dependent on, and return their own event
  - Functors enqueue a kernel—upon return the kernel may not have completed or even begun

Enqueue Instead of Execute

- A function like this

```
this->TCVIntegrate(dt, deltaPos);
```

- Became an enqueue functor call like this

```
if (mIntegratorFunc != NULL)
    event = (*mIntegratorFunc)(event, dt, deltaPos);
```

- The original code remains as a fallback, slightly modified

```
else {
    this->LockBuffers();
    this->TCVIntegrate(dt, deltaPos);
    this->UnlockBuffers();
}
```

Data Model

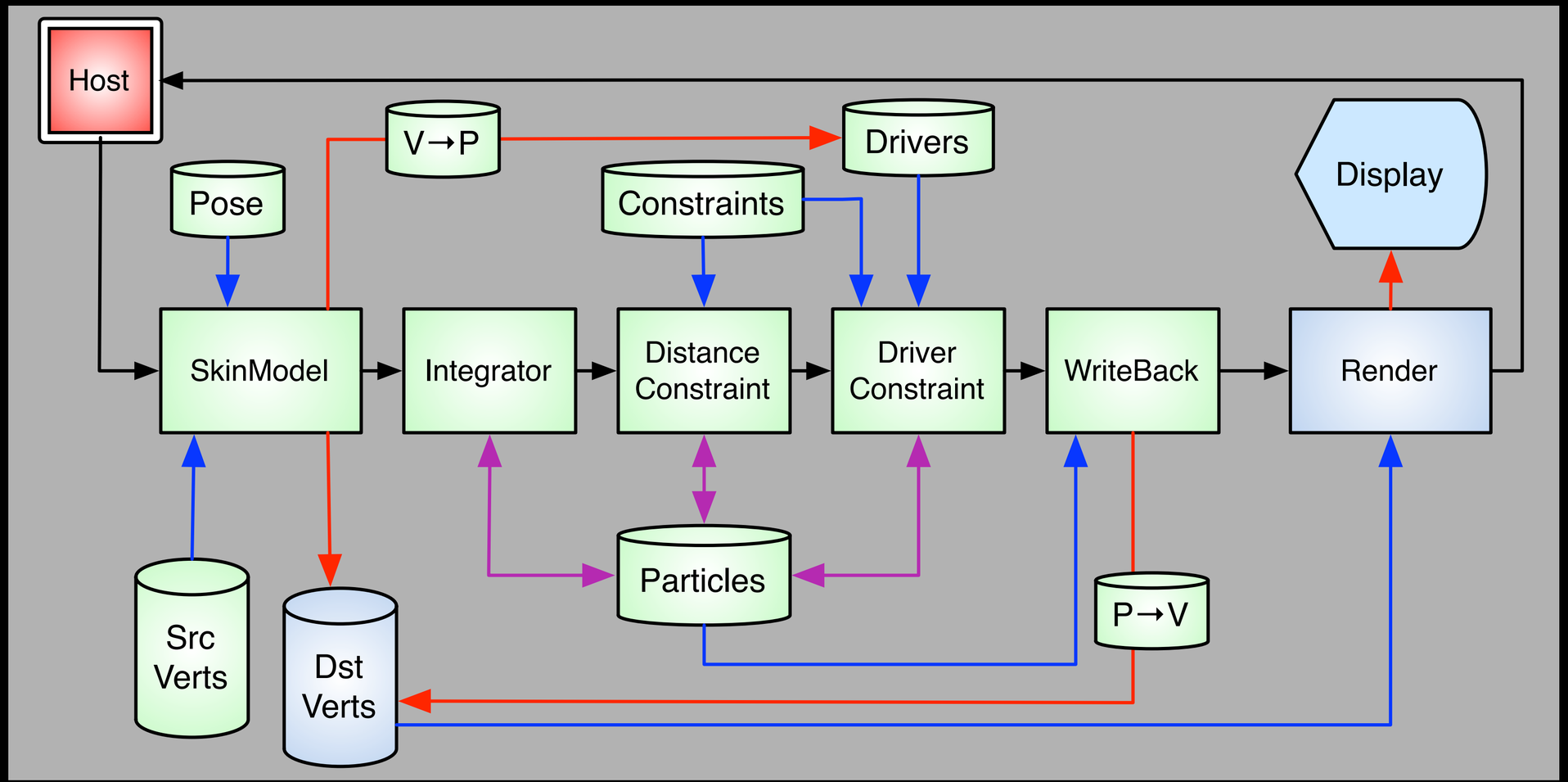
Memory Objects

- Original data was organized in aligned arrays of structs
- `cl_mem` objects were created for each array
  - CL_MEM_USE_HOST_PTR to avoid additional allocations and copying (when using CPU device)
  - OpenGL vertex buffers allocated by GL and bound to OpenCL via GL/CL interop

Demo's Data Buffers

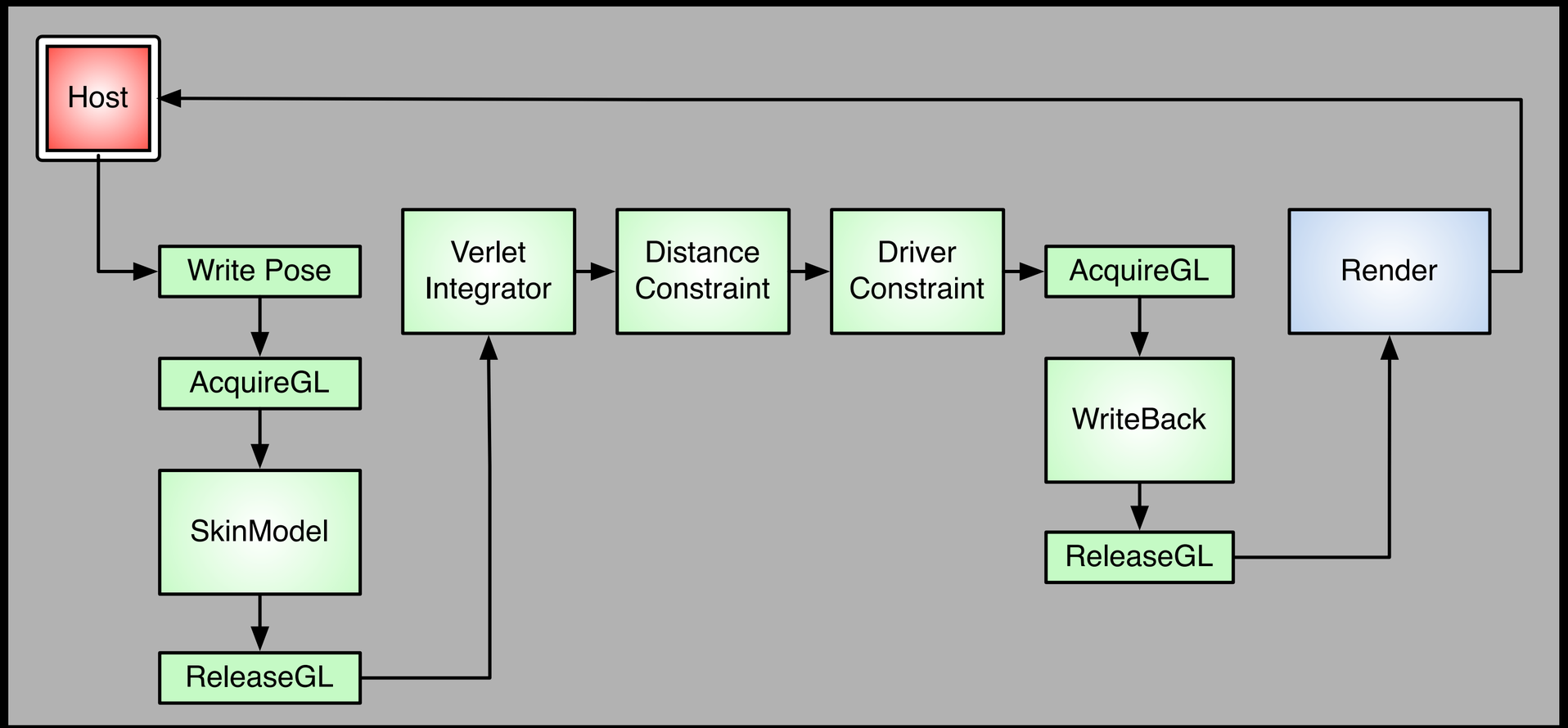
- Output vertices (double buffered, CL/GL buffers)
- Source vertices
- Vertex weights
- Vertex bones
- Skeleton pose
- Particle state
- Drivers (for driver constraint kernel)
- Driver constraints
- Distance constraints
- Vertex→particle mapping
- Particle→vertex mapping

Data Flow Through Kernels



- OpenCL handles buffer locking and interdevice transfer

Complete Command Queue



- Buffer write used to move data into pose buffer
- Acquire/release to give OpenCL access to GL buffers

Kernels

Kernels

- Skinning—Generates vertex buffer and drivers from pose
- Integrator—Acceleration due to gravity
- Distance constraint—Springs between cloth particles
- Driver constraint—Keeps cloth near underlying ‘skin’
- Writeback—Output cloth positions to vertex buffer

Kernels

- Skinning—Generates vertex buffer and drivers from pose
- Integrator—Acceleration due to gravity
- Distance constraint—Springs between cloth particles
- Driver constraint—Keeps cloth near underlying ‘skin’
- Writeback—Output cloth positions to vertex buffer

Kernels

- Skinning—Generates vertex buffer and drivers from pose
- Integrator—Acceleration due to gravity
- Distance constraint—Springs between cloth particles
- Driver constraint—Keeps cloth near underlying ‘skin’
- Writeback—Output cloth positions to vertex buffer

Kernels

- Skinning—Generates vertex buffer and drivers from pose
- Integrator—Acceleration due to gravity
- Distance constraint—Springs between cloth particles
- Driver constraint—Keeps cloth near underlying ‘skin’
- Writeback—Output cloth positions to vertex buffer

Kernels

- Skinning—Generates vertex buffer and drivers from pose
- Integrator—Acceleration due to gravity
- Distance constraint—Springs between cloth particles
- Driver constraint—Keeps cloth near underlying ‘skin’
- Writeback—Output cloth positions to vertex buffer

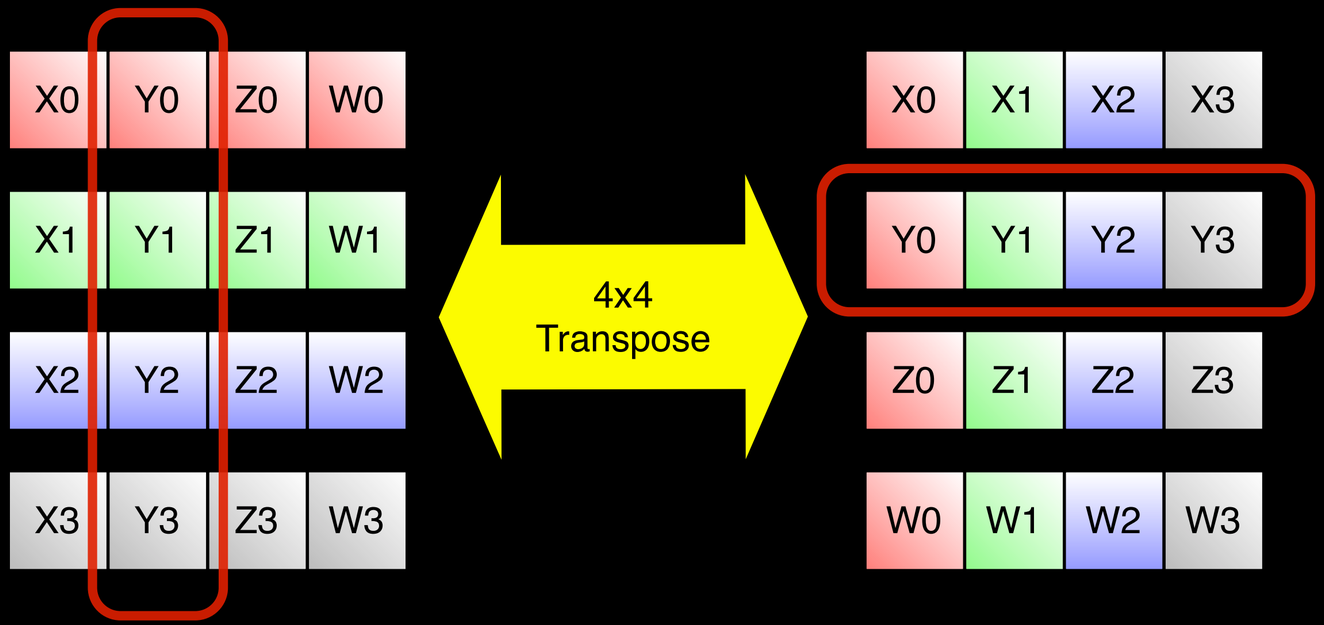
Vectorization

- Wrote two variants of each kernel: Scalar and vector
  - Some hardware does much better with SIMD code

- "Structure of Arrays" (SoA) style math
  - Memory data layout unchanged, rearranged at load/store
  - AoS `float4` contains "xyzw"; SoA `float4` contains "xxxx"

- Two key techniques employed: Transpose and select

4x4 Transpose

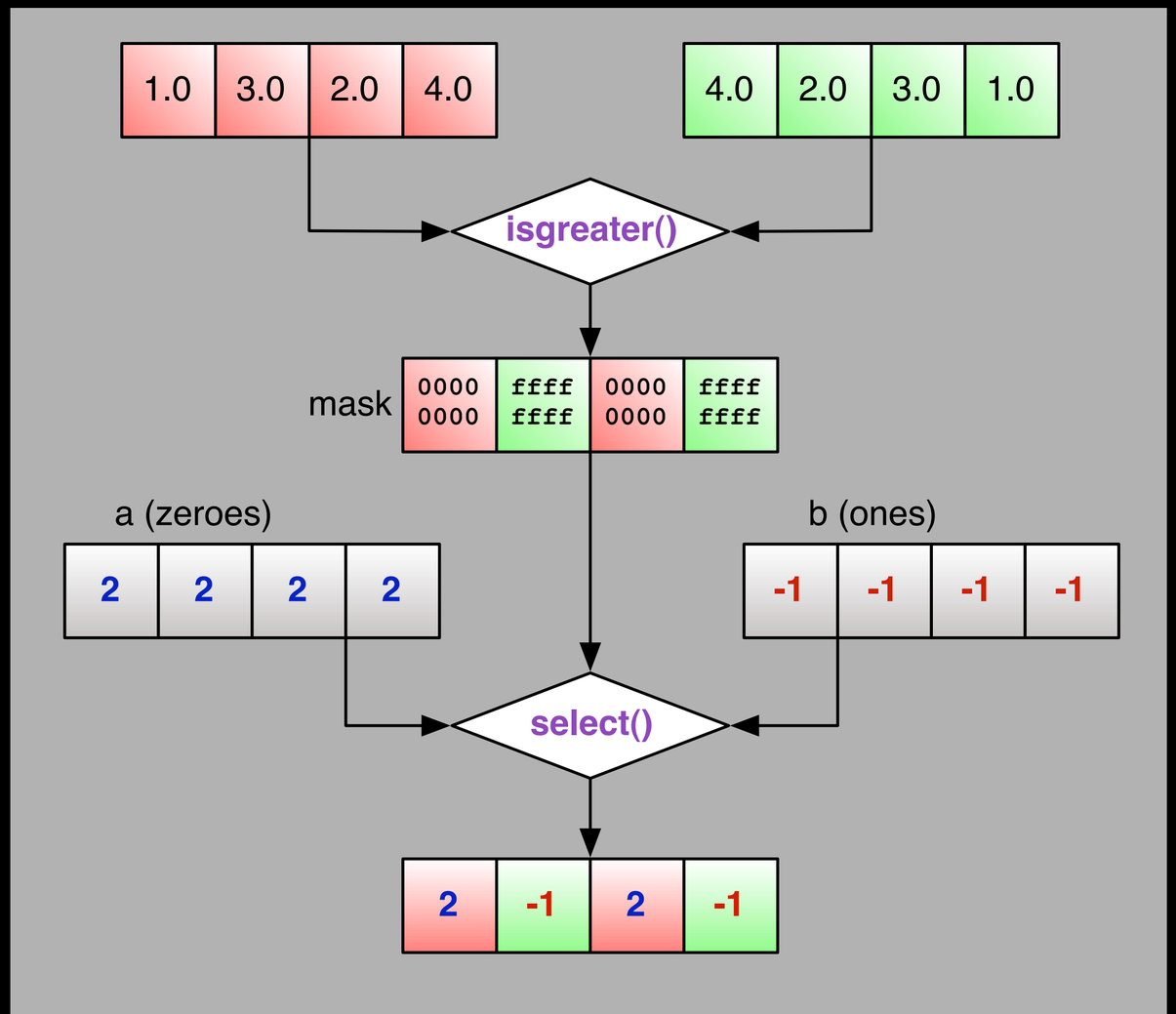



```
float16 transpose (float16 m)
{
  float16 t;
  t.even = m.lo;  t.odd = m.hi;
  m.even = t.lo;  m.odd = t.hi;
  return m;
}
```

- Used to convert between AoS and SoA
- Four `float4` AoS values loaded into one `float16`
- Post transpose the four `float4` parts are SoA

Select vs. Branch

- Branching is bad, for many reasons
- `select(a,b,c)` efficiently chooses between “a” and “b” based on “c” in element-wise fashion
- Comparisons like `isgreater(a,b)` are set up to output “c”



Scalar Integrator

```
  void perform_integrator (int pIdx, float16 deltaPos, float4 gravity_impulse, float dt,
     __global Particle* particles, __global ParticleDebug* particlesDebug, __global short* indices,
     __global IntegratorState *iState, __constant ConstraintTuningParameters *ctp)
{
   float4 curPos, curPrevPos, nextPos;
   curPos = particles[pIdx].mPos;
   curPrevPos = particles[pIdx].mPrevPos;

   float verticalSpeedRatio = (1.0f - ctp->mVerticalSpeedDampening);

   // TIME CORRECTED VERLET
   // xi+1 = xi + (xi - xi-1) * (dti / dti-1) + a * dti * dti
   particlesDebug[pIdx].mColor = tRGBA(1.0f,1.0f,1.0f,1.0f);
   particlesDebug[pIdx].mLength = 0.0f;

   if ((indices[pIdx]) >= 0 && (curPrevPos.w > 0.0f))
   {
       nextPos = curPos;
       nextPos -= curPrevPos;
       nextPos *= dt / iState->mLastDT;
       nextPos.y *= verticalSpeedRatio;
       nextPos += gravity_impulse;

       particles[pIdx].mPrevPos = curPos;
       particles[pIdx].mPos = curPos + nextPos;
   }
}
```

Vector Integrator

```
void perform_vector_integrator (float4 vdt_ratio, float4 acc, int numparticles, int pIdx,
__global Particle *ptrParticle, __global ParticleDebug *ptrParticleDebug, __global uint *mapped)
{
float16 curPos, curPrevPos;  // load particle position and previous position
curPos.s0123 = ptrParticle[0].mPos;    curPrevPos.s0123 = ptrParticle[0].mPrevPos;
curPos.s4567 = ptrParticle[1].mPos;    curPrevPos.s4567 = ptrParticle[1].mPrevPos;
curPos.s89ab = ptrParticle[2].mPos;    curPrevPos.s89ab = ptrParticle[2].mPrevPos;
curPos.scdef = ptrParticle[3].mPos;    curPrevPos.scdef = ptrParticle[3].mPrevPos;

curPos = transpose(curPos);  // transpose Aos -> SoA
curPrevPos = transpose(curPrevPos);

// extract locked flags for particles
uint4 mask = (uint4)isgreater(curPrevPos.scdef, (float4)0.0f);

// TIME CORRECTED VERLET
float4 next_x = ((curPos.s0123 - curPrevPos.s0123) * vdt_ratio + (curPos.s0123 + acc.x));
float4 next_y = ((curPos.s4567 - curPrevPos.s4567) * vdt_ratio + (curPos.s4567 + acc.y));
float4 next_z = ((curPos.s89ab - curPrevPos.s89ab) * vdt_ratio + (curPos.s89ab + acc.z));

// select between unchanged (if locked) and new location (if not locked)
curPrevPos.s0123 = select(curPrevPos.s0123, curPos.s0123, mask);
curPrevPos.s4567 = select(curPrevPos.s4567, curPos.s4567, mask);
curPrevPos.s89ab = select(curPrevPos.s89ab, curPos.s89ab, mask);
curPos.s0123 = select(curPos.s0123, next_x, mask);
curPos.s4567 = select(curPos.s4567, next_y, mask);
curPos.s89ab = select(curPos.s89ab, next_z, mask);

curPos = transpose(curPos);  // transpose SoA -> AoS
curPrevPos = transpose(curPrevPos);

ptrParticle[0].mPos = curPos.s0123;    ptrParticle[0].mPrevPos = curPrevPos.s0123; // store particles back
ptrParticle[1].mPos = curPos.s4567;    ptrParticle[1].mPrevPos = curPrevPos.s4567;
ptrParticle[2].mPos = curPos.s89ab;    ptrParticle[2].mPrevPos = curPrevPos.s89ab;
ptrParticle[3].mPos = curPos.scdef;    ptrParticle[3].mPrevPos = curPrevPos.scdef;
}
```

Vector Integrator

```
void perform_vector_integrator (float4 vdt_ratio, float4 acc, int numparticles, int pIdx,
__global Particle *ptrParticle, __global ParticleDebug *ptrParticleDebug, __global uint *mapped)
{
float16 curPos, curPrevPos;  // load particle position and previous position
curPos.s0123 = ptrParticle[0].mPos;    curPrevPos.s0123 = ptrParticle[0].mPrevPos;
curPos.s4567 = ptrParticle[1].mPos;    curPrevPos.s4567 = ptrParticle[1].mPrevPos;
curPos.s89ab = ptrParticle[2].mPos;    curPrevPos.s89ab = ptrParticle[2].mPrevPos;
curPos.scdef = ptrParticle[3].mPos;    curPrevPos.scdef = ptrParticle[3].mPrevPos;

curPos = transpose(curPos);  // transpose Aos -> SoA
curPrevPos = transpose(curPrevPos);

// extract locked flags for particles
uint4 mask = (uint4)isgreater(curPrevPos.scdef, (float4)0.0f);

// TIME CORRECTED VERLET
float4 next_x = ((curPos.s0123 - curPrevPos.s0123) * vdt_ratio + (curPos.s0123 + acc.x));
float4 next_y = ((curPos.s4567 - curPrevPos.s4567) * vdt_ratio + (curPos.s4567 + acc.y));
float4 next_z = ((curPos.s89ab - curPrevPos.s89ab) * vdt_ratio + (curPos.s89ab + acc.z));

// select between unchanged (if locked) and new location (if not locked)
curPrevPos.s0123 = select(curPrevPos.s0123, curPos.s0123, mask);
curPrevPos.s4567 = select(curPrevPos.s4567, curPos.s4567, mask);
curPrevPos.s89ab = select(curPrevPos.s89ab, curPos.s89ab, mask);
curPos.s0123 = select(curPos.s0123, next_x, mask);
curPos.s4567 = select(curPos.s4567, next_y, mask);
curPos.s89ab = select(curPos.s89ab, next_z, mask);

curPos = transpose(curPos);  // transpose SoA -> AoS
curPrevPos = transpose(curPrevPos);

ptrParticle[0].mPos = curPos.s0123;    ptrParticle[0].mPrevPos = curPrevPos.s0123; // store particles back
ptrParticle[1].mPos = curPos.s4567;    ptrParticle[1].mPrevPos = curPrevPos.s4567;
ptrParticle[2].mPos = curPos.s89ab;    ptrParticle[2].mPrevPos = curPrevPos.s89ab;
ptrParticle[3].mPos = curPos.scdef;    ptrParticle[3].mPrevPos = curPrevPos.scdef;
}
```

Transpose

Vector Integrator

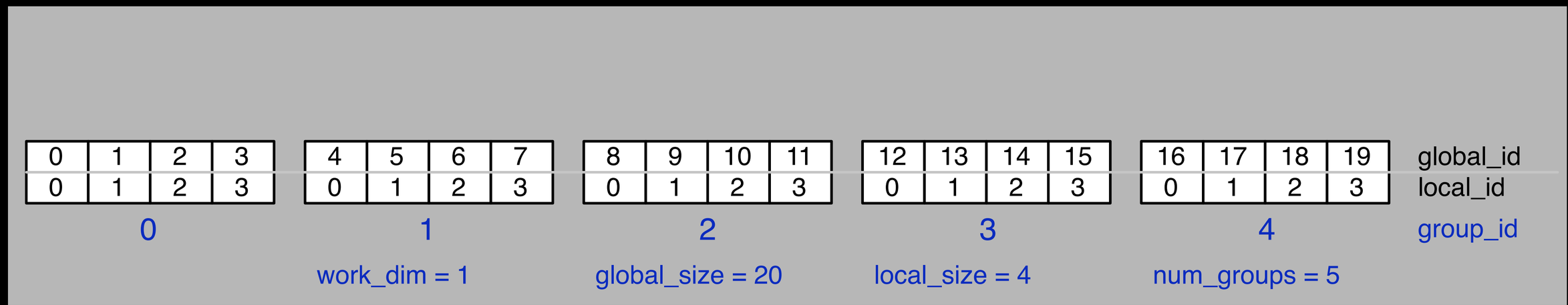
Compare and Selects

```
void perform_vector_integrator (float4 vdt_ratio, float4 acc, int numparticles, int pIdx,
__global Particle *ptrParticle, __global ParticleDebug *ptrParticleDebug, __global uint *mapped)
{
float16 curPos, curPrevPos;  // load particle position and previous position
curPos.s0123 = ptrParticle[0].mPos;    curPrevPos.s0123 = ptrParticle[0].mPrevPos;
curPos.s4567 = ptrParticle[1].mPos;    curPrevPos.s4567 = ptrParticle[1].mPrevPos;
curPos.s89ab = ptrParticle[2].mPos;    curPrevPos.s89ab = ptrParticle[2].mPrevPos;
curPos.scdef = ptrParticle[3].mPos;    curPrevPos.scdef = ptrParticle[3].mPrevPos;

curPos = transpose(curPos);  // transpose Aos -> SoA
curPrevPos = transpose(curPrevPos);

// extract locked flags for particles
uint4 mask = (uint4)isgreater(curPrevPos.scdef, (float4)0.0f);

// TIME CORRECTED VERLET
float4 next_x = ((curPos.s0123 - curPrevPos.s0123) * vdt_ratio + (curPos.s0123 + acc.x));
float4 next_y = ((curPos.s4567 - curPrevPos.s4567) * vdt_ratio + (curPos.s4567 + acc.y));
float4 next_z = ((curPos.s89ab - curPrevPos.s89ab) * vdt_ratio + (curPos.s89ab + acc.z));

// select between unchanged (if locked) and new location (if not locked)
curPrevPos.s0123 = select(curPrevPos.s0123, curPos.s0123, mask);
curPrevPos.s4567 = select(curPrevPos.s4567, curPos.s4567, mask);
curPrevPos.s89ab = select(curPrevPos.s89ab, curPos.s89ab, mask);
curPos.s0123 = select(curPos.s0123, next_x, mask);
curPos.s4567 = select(curPos.s4567, next_y, mask);
curPos.s89ab = select(curPos.s89ab, next_z, mask);

curPos = transpose(curPos);  // transpose SoA -> AoS
curPrevPos = transpose(curPrevPos);

ptrParticle[0].mPos = curPos.s0123;    ptrParticle[0].mPrevPos = curPrevPos.s0123; // store particles back
ptrParticle[1].mPos = curPos.s4567;    ptrParticle[1].mPrevPos = curPrevPos.s4567;
ptrParticle[2].mPos = curPos.s89ab;    ptrParticle[2].mPrevPos = curPrevPos.s89ab;
ptrParticle[3].mPos = curPos.scdef;    ptrParticle[3].mPrevPos = curPrevPos.scdef;
}
```

Concurrency

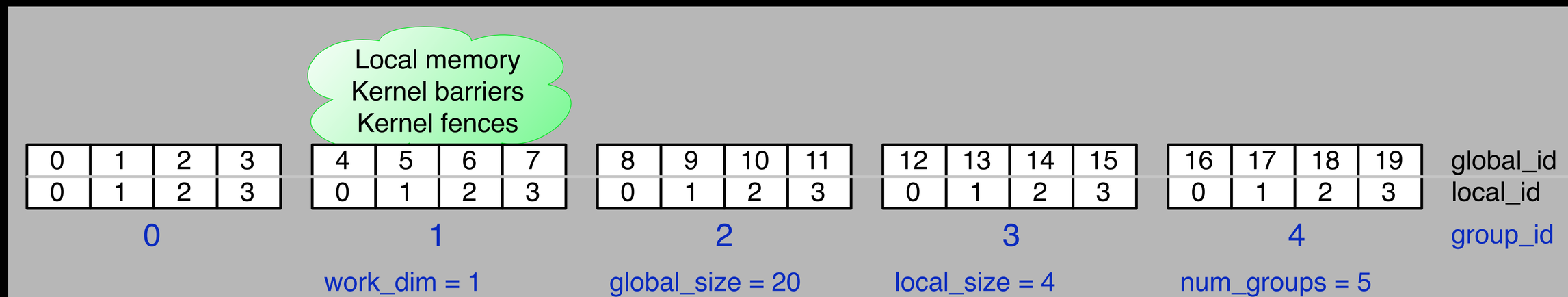
Work-Items

- Data parallel model in OpenCL is based on work-items
  - Each work-item is given its own index (in up to three dimensions)
  - Work-items are organized into uniformly sized “workgroups”
  - Workgroup size is limited by device and kernel



Workgroups

- Work-items in a workgroup execute in parallel and share
  - Local memory
  - Barriers
  - Fences



Per Kernel Index Space

- This demo's kernels are all in one-dimensional spaces
- Each kernel uses its own space
  - Skinning: Complete vertex array
  - Integrator: Particle array
  - Driver constraint: Driver array
  - Distance constraint: Constraints array
  - Writeback: Clothed portion of vertex array

Distance Constraint: Limited Parallelism

- Each distance constraint modifies two particles
  - Modifying a particle from multiple constraints concurrently gives incorrect results
- Original constraints organized into “octets”
  - Eight-at-once is far too few to keep GPU busy

Performance

CPU Device Performance

	Host	Scalar Task	Scalar DP (8 Cores)	Vector DP (8 Cores)
Overall	1.0	2.72	17.03	17.27
Skinning	1.0	2.98	20.98	20.98*
Integrator	1.0	1.53	1.48	1.10
Distance	1.0	1.34	2.54	2.69
Driver	1.0	1.14	5.58	8.83
WriteBack	1.0	1.01	7.26	7.26*

*No vector version available

GPU Device Performance

	Host	Best CPU (8 Cores)	Unoptimized GTX285	
Overall	1.0	17.27	4.11	
Skinning	1.0	20.98	4.39	
Integrator	1.0	1.53	1.10	
Distance	1.0	2.69	0.74	
Driver	1.0	8.83	3.24	
WriteBack	1.0	7.26	17.69	

Optimizations—Algorithmic

- Distance constraint's original limit of eight simultaneous work-items is crippling to GPU performance
- Reordering constraints to maximize the workgroup size (7x) improved kernel performance dramatically
- Limited by algorithm, alternatives should be investigated

Optimizations—Work per Task

- Larger simultaneous data set sizes provide the GPU with substantially more work
  - CPU benefits as well, but drops off with larger data sets
- Real game: Process all characters as a single batch
- Demo: Simple geometry replication (25x)

Optimizations—Bandwidth

- Memory access is crucial in bandwidth limited kernels (and most will be bandwidth limited)

- GL/CL integration
  - CL_MEM_COPY_HOST_PTR for GPU
  - Current GPU memory controllers are optimized for a specific set of memory access patterns

GPU Memory and Execution Optimization

- Burst reads from memory are essential
- Bursts only happen on sequential accesses
- Bursts are created by coalescing smaller reads
- Can only coalesce 4-, 8-, or 16-byte reads
- Coalesces across work-items in a workgroup

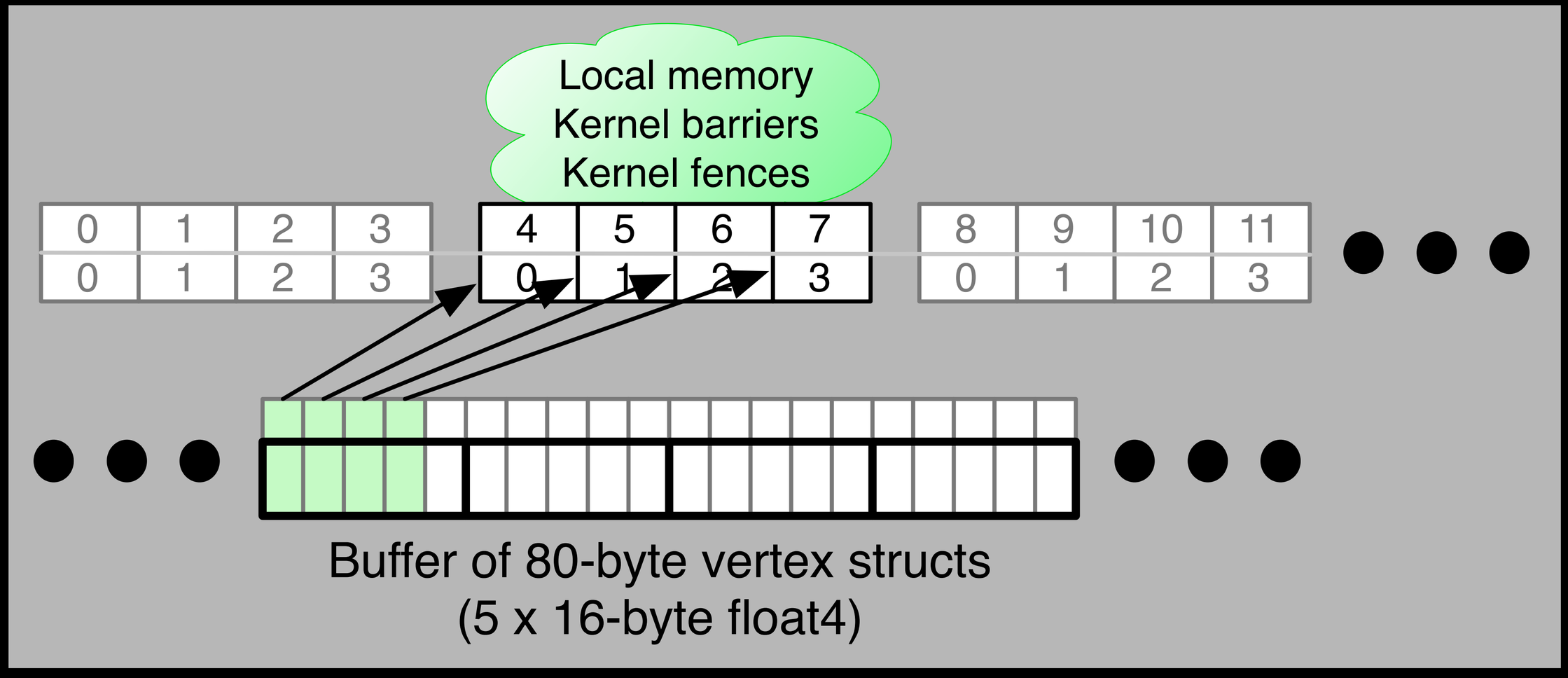
GPU Memory and Execution Optimization

- This pattern must currently be explicitly coded for
- AoS access from local memory can be done efficiently

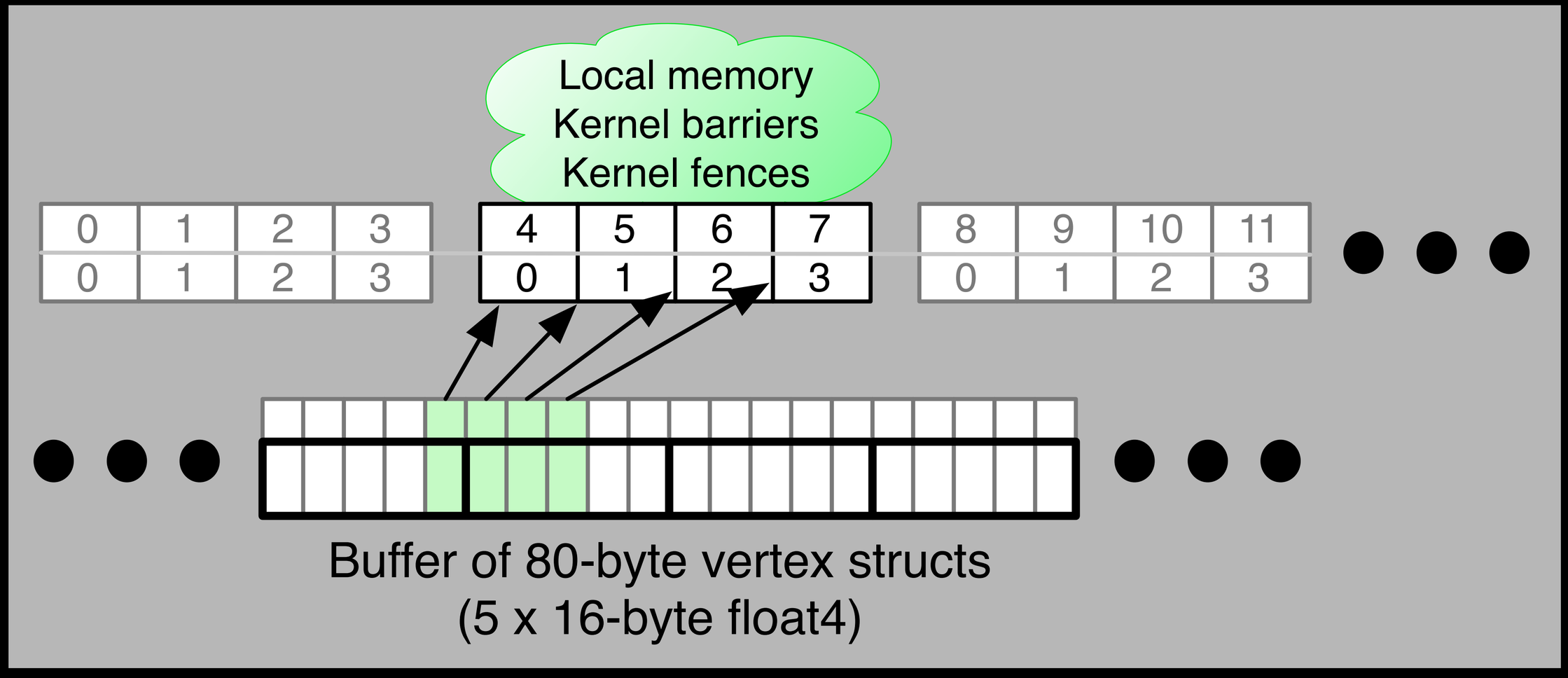
- Solution: Optimized transfer from global to local memory, then operate on local memory

- Newest GPUs do substantially better
- There may be possible efficiency improvements to the compiler in the future to help address this

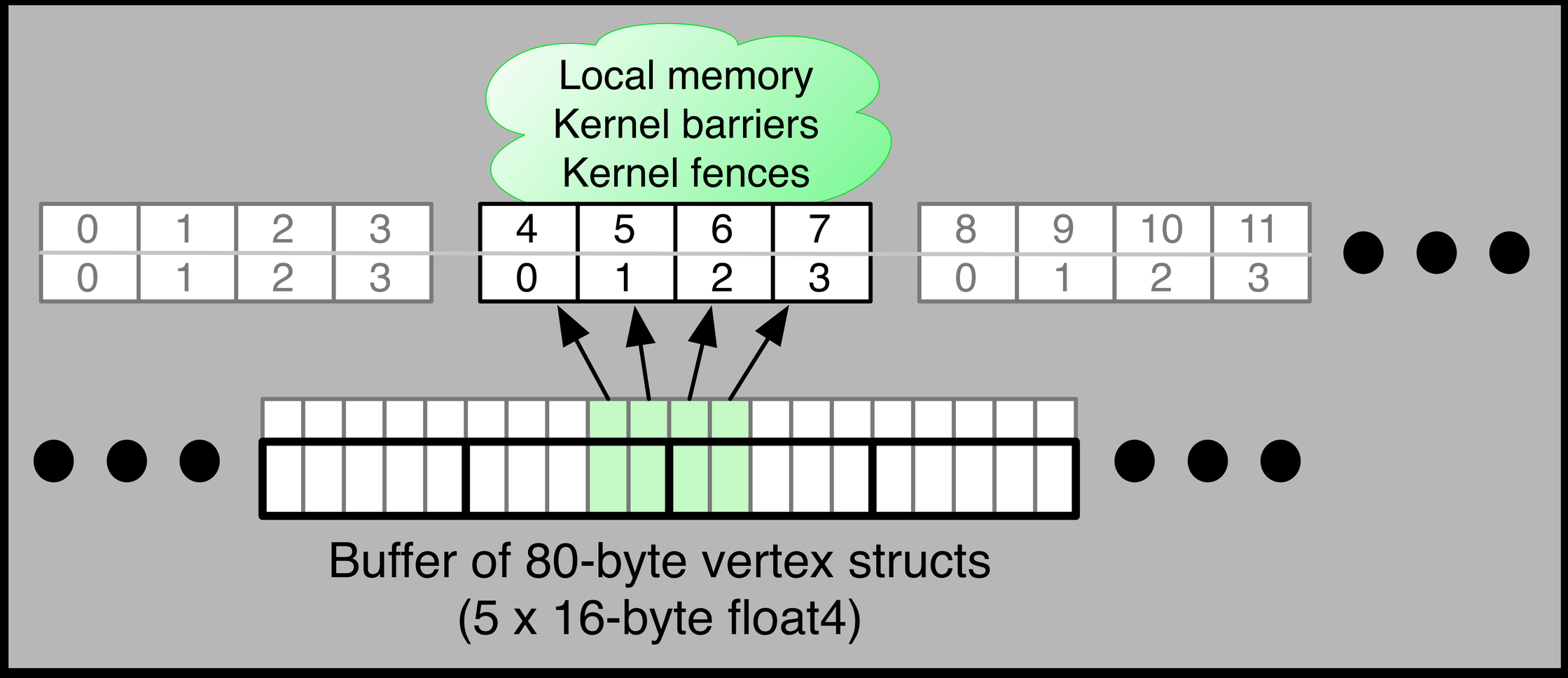
GPU Memory and Execution Optimization



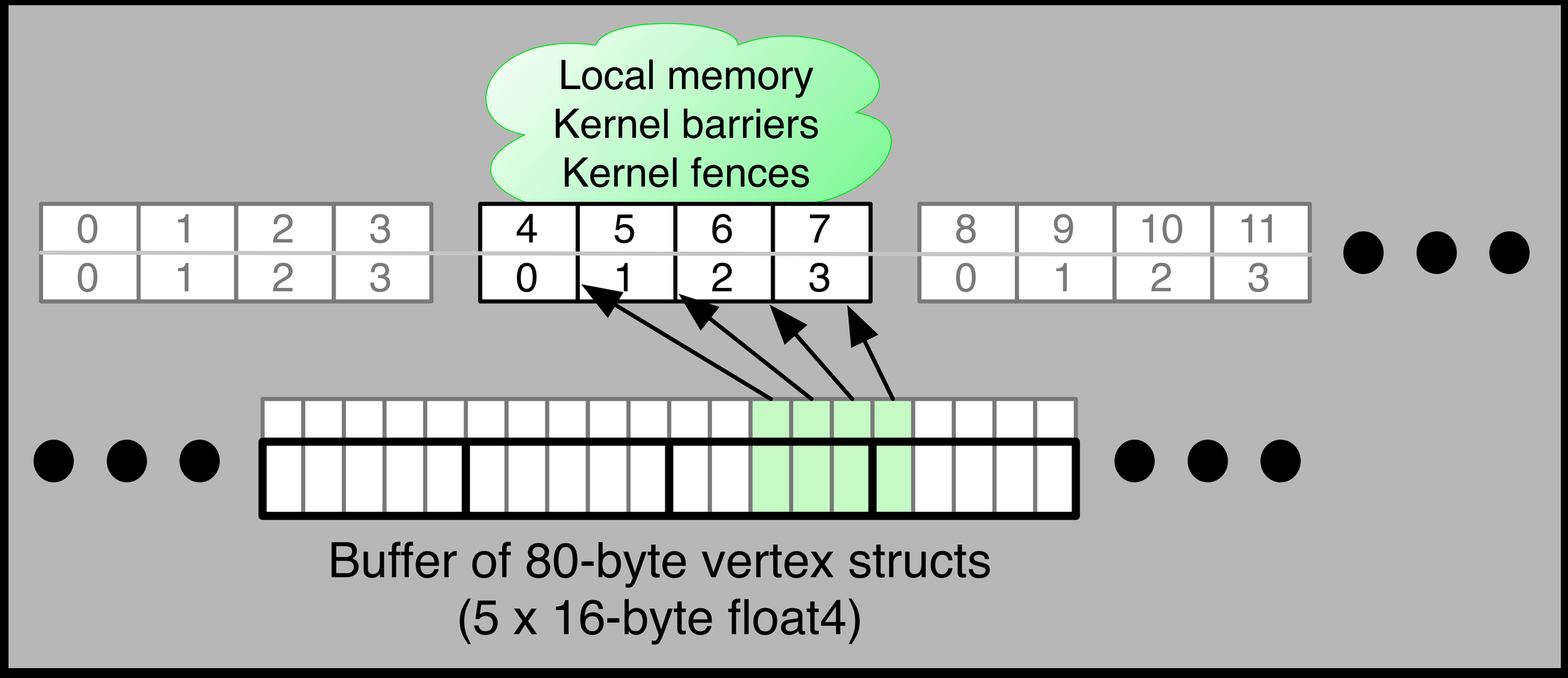
GPU Memory and Execution Optimization



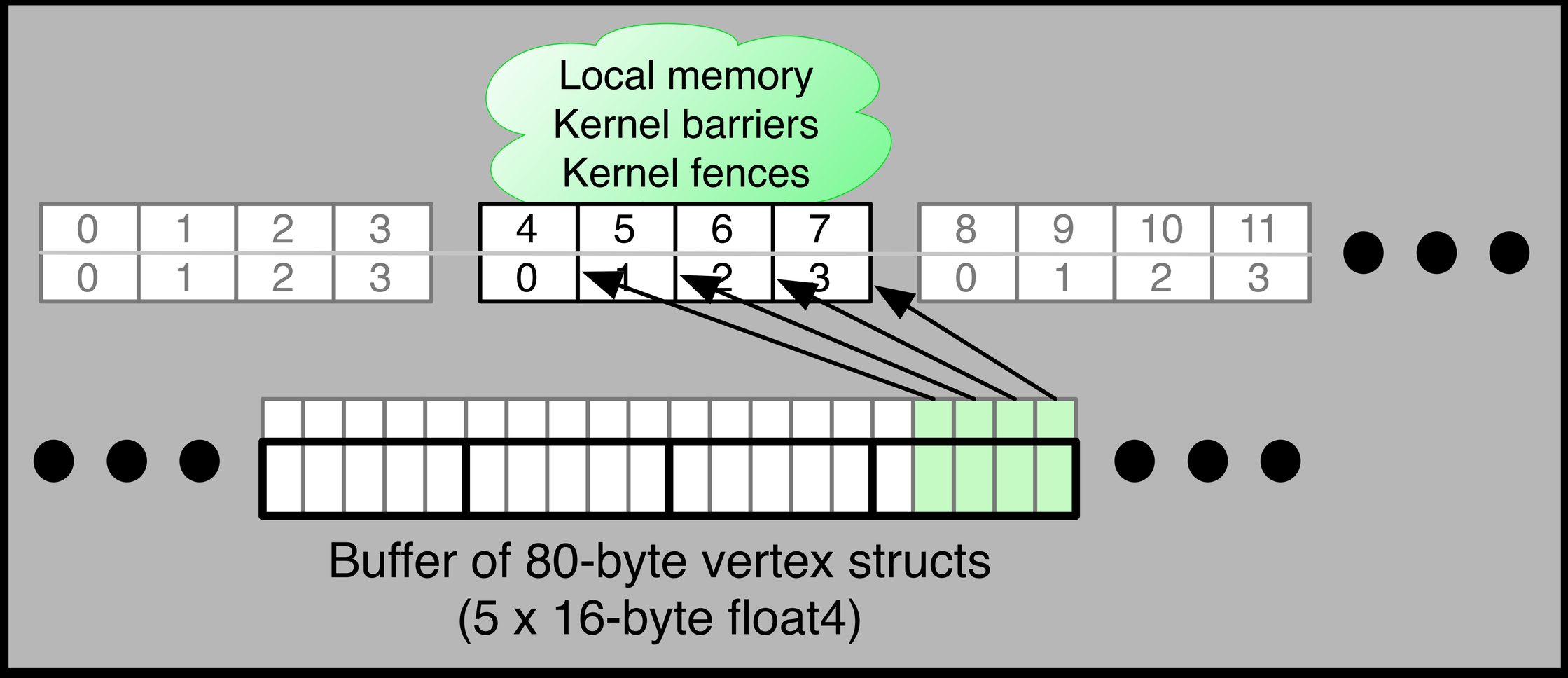
GPU Memory and Execution Optimization



GPU Memory and Execution Optimization



GPU Memory and Execution Optimization

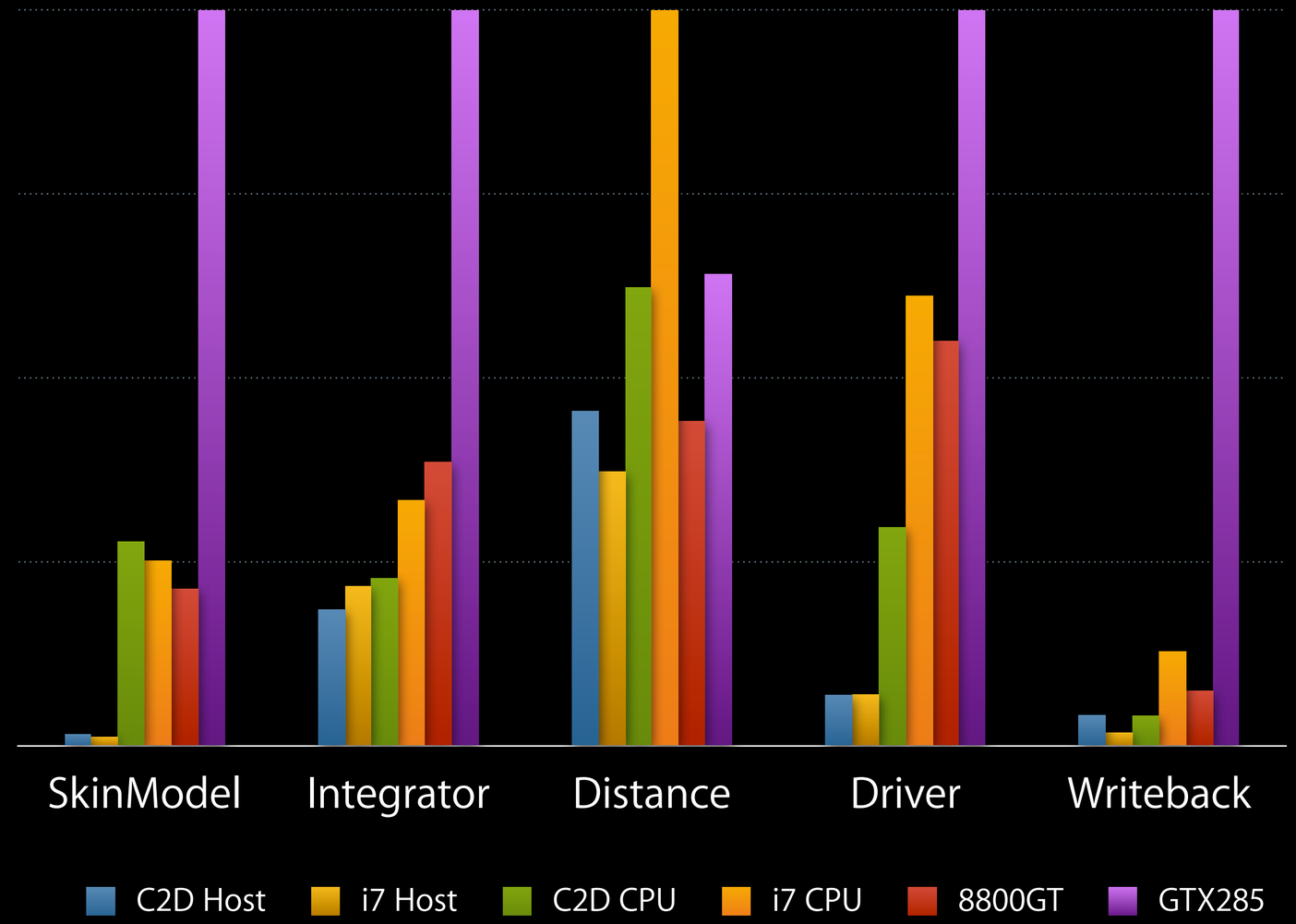


GPU Device Performance

	Host	Best CPU (8 Cores)	Unoptimized GTX285	Optimized GTX285
Overall	1.0	17.27	4.11	45.15
Skinning	1.0	20.98	4.39	83.56
Integrator	1.0	1.53	1.10	4.6
Distance	1.0	2.69	0.74	1.73
Driver	1.0	8.83	3.24	14.45
WriteBack	1.0	7.26	17.69	56.60

Performance Summary

- Performance tuning is essential
- Performance characteristics are not hidden by abstraction layer
- Expect to write multiple kernel variations and choose empirically at runtime



More Information

Allan Schaffer
2D and 3D Graphics Technologies Evangelist
aschaffer@apple.com

Q&A

WWDC Session Survey

You can rate your sessions on the WWDC Attendee Site.

Survey forms are located on each session description page at:

developer.apple.com/wwdc/attendee

WWDC09